Object-Oriented Programming

- a programming paradigm based on the concept of **objects**.
- classes are often used, but not required (JS)
- objects talk to each other with "messages" (method calls)
- each object has hidden implementation (code) and internal state (data)

Essential Components of OOP

- *Classes*: a blueprint for creating new objects – it defines a public interface, code for methods, and data fields.
- *Interfaces*: a related group of function prototypes that we want one or more classes to implement.
- *Objects*: represent a particular "thing" - like a circle of radius 5 at location (1,3). Each has its own interface, code, and field values.
- *Inheritance*: A derived class inherits either the code, the interface, or both from a base class. The derived class can override the base class's code or add to its interface.
- *Subtype Polymorphism*: Code designed to operate on an object of type T (e.g., Person) can operate on any object that is a subtype of T (e.g., Student).
- *Dynamic Dispatch*: The actual code that runs when a method is called depends on the target object, and can only be determined at runtime.
- *Encapsulation*: Encapsulation is the bundling of a public interface and private data fields/code together into a cohesive unit.

Classes

Key elements: fields (member variables), the public interface (public methods), private methods, and method implementations.

Every time you define a class you define a new type

- when you define a class, it also defines a new type
- if the class is a concrete class (all of its methods have { definitions }) and not an *abstract class* (missing one or more method { implementations }) then you can use this new type to define all of the following: objects, object references/pointers, and references
- A class is a class, and a type is a type; they're different

Objects

- An object is a distinct value, each object has its own copy of fields and methods
- generally, objects *instantiated* by using a class
- A class serves as a blueprint for creating new objects
  - but not always! In some languages, like JavaScript, there are no classes, only objects! (JS "classes" are syntactic sugar)

Interfaces

- a related group of **function prototypes** (declarations without { bodies }) that describes behaviors that we want one or more classes to implement
- specifies what we want one or more classes to do, but not how to implement these interfaces
- can have a class to *implement* or *inherit* from an interface
- interfaces enable otherwise-unrelated classes to provide a common set of behaviors without actually inheriting any code

```java
// Java interface definition for shapes
interface Shape {
  public double area();
  public double perimeter();
}

class Square implements Shape {
 public double area()
   { return w_ * w_; }
 public double perimeter()
   { return 4 * w_; }
}
```

Every time you define an interface you define a new reference type

- Since an interface lacks any function { definitions/code }, you can't use a reference type to define full objects:
- But you can use reference types to define object references, pointers and references (depending on what the language supports)

```java
Shape x = new Shape();  // allocating a "new Shape()" doesn't work!
void someFunc(Shape s) { ... }  // s is an object reference; this works
```

- Each interface defines a type, and each class defines a type, and a class implements an interface – so the class has 2 types!
- Ex: Square class is of the Square type AND it is of the Shape type!
  - you can pass a Square to any function that accepts either a Square or a Shape object reference, pointer or reference!
- when you define a class with all of its methods { implemented }, you create a "value type" or a concrete type - and you may define all of the following: objects, object references/pointers, and references.

Classes, Interfaces, Object Challenges

Q: Why even support the notion of a class in a language? What are the pros and cons of using a model like JavaScript with just objects?

A: Classes enable us to create a consistent set of objects, all generated in a consistent way based on the class specification. Classes enable us to define a new type, which we'll see is useful for polymorphism in statically typed languages

Q: JavaScript can dynamically add methods to objects, is there any reason we can't dynamically add/remove methods to/from classes?

While most/all static languages don't allow this, dynamically typed languages absolutely allow this. In a statically typed language, adding a new method would potentially change the class interface which would complicate compilation/interpretation, but technically this could be done too.

Encapsulation

Encapsulation is the guiding principle behind the OOP paradigm. There are two facets to encapsulation:

- bundle related *public interface*, *data*, and *code* together into a cohesive, problem-solving machine.
- *hide data/implementation* details of that machine from its clients – forcing them to use its public interface.

Benefits:

- Simpler programs: reduced coupling between modules, simplifying our programs
- Easier Improvements: can improve implementations without impacting other components
- Better Modularity: can build a class once and re-use it in different contexts

One reason for OOP's success is that it helps people build software that is resilient to many types of change.

Shallow-dive Into Classes

This and Self

- when a method is called, it must know what object it is to operate on.
- in most languages, this is done transparently when you call a method. the language passes a reference to as a hidden extra argument, and it may be referenced in the method as `self` or `this`
- `self` / `this` is an object reference or pointer in most languages.
  - in Python `self` is required. Without explicit member variable declarations, assignment without `self` would be ambiguous

Here's an illustration of how *this* is implemented in c++:

```
// The code you write:
class Nerd {
 private:
   string name;
   int IQ;

 public:
   Nerd(const string& name) {
     this->name = name;
     IQ = 100; // this->IQ is optional
   }

   string talk() {
     return this->name + " likes PI.";
   }
};

int main() {
 Nerd n("Carey");
 cout << n.talk();
}
```

```
// What's actually happening for the code above:
struct Nerd {
  string name;
  int IQ;
};

void init(Nerd *this, const string& name) {
  this->name = name;
  this->IQ = 100;
}

string talk(Nerd *this) {
  return this->name + " likes PI.";
}

int main() {
 Nerd n;
 init(&n, "Carey");
 cout << talk(&n);
}
```

Access Modifiers

- A key goal of encapsulation is to hide the data/implementation details from users of a class
- Languages provide "access modifier" syntax to enable a class to designate what parts of a class are publicly visible and to hide the class's data/implementation.
  - *public*: may be accessed by any part of your program
  - *protected*: may be accessed by C or any subclass derived from C
  - *private*: may only be accessed by methods in class C
- Public, protected and private generally have the same semantics across languages, though there are exceptions

Python uses underscores to indicate protected/private methods (though protected methods aren't enforced, private are though):

```
# Python access modifiers
class Person:
 def __init__(self, name):
   self.__name = name

 def talk(self):         # Public method
   print("Hi, my name is "+ self.__name)

 def _daydream(self):    # Protected method, not enforced
   print("Rainbow narwhals")

 def __poop(self):       # Private method
   print("Grunt... plop!")


# Derived class
class Student(Person):
 ...
 def do_stuff(self):
   self.talk()           # Works fine - public
   self._daydream()      # Works fine - protected
   self.__poop()

a = Student("Cindi")
a.talk()          # Works fine - public
a._daydream()     # works!! - Python doesn't enforce protected methods
a.__poop()        # Generates error
```

Encapsulation Best Practices

- Design your classes to hide all implementation details from other classes
- Make all member fields, constants, and helper methods private
- Avoid providing getters/setters for fields that are specific to your current implementation, to reduce coupling with other classes
- constructors should completely initialize objects, so users don't have to call other methods to create a fully-valid object

Properties, Accessors and Mutators

sometimes want to expose a field (aka property) of an object for external use

- **property** - some attribute of an object
- **accessor** - method that gets the value of such a property.
- **mutator** - method that changes the value of a property.

many languages have syntax to define accessors and mutators more easily.

- accessors and mutators for a property enable us to hide a field's implementation and more easily refactor
- can define a function which abstracts the underlying implementation of a given field/property
- Some languages actually support built-in syntax to create accessors/mutators for properties

```
# Python's accessors/mutators
class Person:
 ...

 @property
 def age(self):      # accessor
   return self._age

 @age.setter
 def age(self, age):  # mutator
   if age >= 0 and age <= 130:
     self._age = age
   else:
     raise ValueError("Invalid age!")

a = Person("Archna", 22)
print(a.age)
a.age = 23
```

- `@property` indicates accessor. Simply referring to the function by name, just like you would a member variable - causes it to be called (`a.age`)
- `@age.setter` designates that the second age method (which takes two parameters) is a mutator. Simply using assignment, causes this method to be called and the value of 23 to be passed in as the age argument.

Property Best Practices

Property vs. defining a traditional method (in a language with explicit support/syntax for properties)?

Rule of thumb:

- use properties when you're exposing the state of a class (even if exposing that state requires minor computation)
- use traditional methods when you're exposing behaviors of a class or exposing a state that requires heavy computation (e.g., a database query that might take tens of milliseconds or more)
- in general, don't use properties to expose implementation details of a class that might change if you update your implementation
  - exposing a property for some internal member variable that is specific to your current implementation is bad

Inheritance

- **Interface Inheritance**: creating many classes that share a common public base interface `I` . A class implements interface `I` by providing implementations for all `I` 's functions.
- **Implementation Inheritance**: reusing a base class's method implementations. A derived class inherits method implementations from a base class.
- **Subtype Inheritance**: base class provides a public interface and implementations for its methods. A derived class inherits both the base class's interface and its implementations.
  - mix of interface and implementation inheritance
- **Prototypical Inheritance**: One object may inherit the fields/methods from a parent object. This is used in JavaScript.

Interface Inheritance

- interface - a collection of function prototypes (aka declarations) that provide a common set of behaviors, but there's no implementations to inherit

```
class Washable { // interface
public:
  virtual void wash() = 0;
  virtual void dry() = 0;
};

class Car: public Washable {
public:
  void drive() { cout << "Vroom vroom"; }
  void wash() { cout << "Use soap"; }
  void dry()  { cout << "Use rag"; }
};
```

```
class Person: public Washable {
public:
  void talk() { cout << "Hello!"; }
  void wash() { cout << "Use shampoo"; }
  void dry()  { cout << "Use blowdryer"; }
};
```

- with interface inheritance we can also have a single class support multiple interfaces

```
class Washable {
public:
  virtual void wash() = 0;
};

class Drivable {
public:
  virtual void accelerate() = 0;
};

class Car: public Washable, public Drivable {
public:
  ...

  // Washable methods
  void wash() { cout << "Use soap"; }

  // Drivable methods
  void accelerate() { cout << "Vroom vroom"; }
};
```

- classes that implement interfaces have *multiple personas/types*: they are the class type and the interface type
  - `Car` has 3 personas: Car, Drivable, Washable

```
// can use the interface as a reference type, so now w can refer to any washable
object
void cleanUp(Washable& w) {
  w.wash();
}

// or any drivable object
void driveToUCLA(Drivable& d) {
 d.accelerate();
}

int main() {
  Car forrester;
  cleanUp(forrester);
  driveToUCLA(forrester);
}
```

Go (and Rust) uses **structural inheritance**, where you don't need to explicitly say that a `Circle` implements a `Shape`; all that needs to be true is that it implements the relevant functions.

```
type Shape interface {
  Area() float64
  Perimeter() float64
}

type Circle struct {
  x      float64
  y      float64
  radius float64
}

func (c Circle) Area() float64 {
    return PI * c.radius * c.radius }
func (c Circle) Perimeter() float64 {
    return 2 * PI * c.radius }
```

Common Use Cases

1. comparing objects - ex Java's `Comparable`
2. iterating oer an object - ex Java's `Iterable`
3. serializing an object!

Java's `Comparable`:

```
public interface Comparable<T>
{
  public int compareTo(T o);  // Compares two objects, e.g., for sorting
}

public class Dog implements Comparable<Dog> {
  private int bark;
  private int bite;
    ...
  public int compareTo(Dog other) {
    int diff = this.bark – other.bark;
    if (diff != 0) return diff;
    return this.bite – other.bite;
  }
}

ArrayList<Dog> doggos = new ArrayList<>();
...

doggos.sort(Comparator.naturalOrder()); // sort() uses the Comparable interface to
compare objects
```

`iterable`:

```
public interface Iterable<T>
{
  Iterator<T> iterator();
}

public class Dog { ... }

class Kennel implements Iterable<Dog> {
    private ArrayList<Dog> list;
    ...

    public Iterator<Dog> iterator()
      { return list.iterator(); }
}

Kennel kennel = new Kennel();
kennel.addDog(new Dog("Fido",...));
kennel.addDog(new Dog("Nellie",...));

for (Dog d : kennel) {  // for loop uses the Iterable interface
  d.bark();
}
```

Best Practices (Pros & Cons)

Use interface inheritance when

- you have a "can-support" relationship between a class and a group of behaviors.
    - the Car class can support washing.
- you have different classes that all need to support related behaviors, but aren't related to the same base class.
    - Cars and Dogs can both be washed, but aren't related by a common base.

Pros:

- can write functions focused on an interface, and have it apply to many different classes.
- A single class can implement multiple interfaces and play different roles

Cons:

- Doesn't facilitate code reuse

Subclassing Inheritance

- base class `B`, which provides a public interface and method implementations
- derived class `D` which inherits `B`'s public interface and its method implementations
- Derived class `D` may override `B`'s methods, or add its own new methods, potentially expanding `D`'s public interface.

```
// C++ subclassing example
class Shape {
public:
  Shape(float x, float y)  { x_ = x; y_ = y; }
  void setColor(Color c)   { color_ = c; }
  virtual void disappear() { setColor(Black); }  //virtual functions may be overidden
  virtual float area() const = 0;
...
protected:
  void printShapeDebugInfo() const { ... }
};

class Circle: public Shape {
public:
  Circle(float x, float y, float r) :  Shape(x,y)
      { rad_ = r; }
  float radius() const { return rad_; }
  virtual float area() const { return PI * rad_ * rad_; }
  virtual void disappear() {
    for (int i=0;i<10;++i)
        rad_ *= .1;
        Shape::disappear();
  }
};
```

- the derived class, `Circle`, not inherits the public interface of Shape, but also the implementation/code of some of these functions, and the protected functions.
- derived class's interface is expanded, so it can also do its own specialized derived things

Subtype Inheritance Best Practices

Use subtype inheritance when:

- there's an *is-a relationship*:
    - A Circle is a type of Shape
- you expect your subclass to share the entire public interface of the superclass AND maintain the semantics of the super's methods
- you can factor out common implementations from subclasses into a superclass:
    - all Shapes share coordinates and a color along with getters/setters

Don't use when:

- derived class doesn't support the full interface of the base class.
- derived class doesn't share the same semantics as the base class.

```
// C++ Collection class
class Collection {
public:
  void insert(int x) { ... }
  bool delete(int x) { ... }
  void removeDuplicates(int x) { ... }
  int count(int x) const { ... }
  ...

private:
  int *arr_;
  int length_;
};
```

```
// C++ Set class
class Set: public Collection {
public:
  void insert(int x) { ... }
  bool delete(int x) { ... }
  bool contains(int x) const { ... }
  ...

private:
  int *arr_;
  int length_;
};
```

- in this case, you want to inherit the functionality of a base class but don't want to inherit the full public interface (`removeDuplicates` , `count`)
- instead use composition and delegation
- **Composition** - when you make the original class a member variable of your new class
- **Delegation** - when you call the original class's methods from the methods of the new class
- allows you to hide the public interface of the original class (Collection) but the second class can leverage all of its functionality.

```
class Set
public:
  void insert(int x) {
    if (c_.howMany() == 0) c_.insert();   // delegation
  }
  bool delete(int x) {
    return c_.delete(x)
  }
  bool contains(int x) const {
    return c_.count(x) == 1;
  }
  ...
private:
  Collection c_;    // composition
};
```

Fragility

- when changes in a base class break derived class

```
class Container {
public:
  virtual bool contains(int x) const { ... }
  virtual void insert(int x) { v_.push_back(x); }
  virtual void insertAll(vector<int>& items) {
    for(const int x: items)
      // insert(x);
      v_.push_back(x);  // now there is no duplicate check!
  }
private:
  vector<int> v_;
};

// A set holds only unique items
class Set: public Container {
public:
  virtual void insert(int x) {
    if (!contains(x)) Container::insert(x);
  }
};
```

Pros

- Eliminates code duplication/facilitates code reuse
- Simpler maintenance – fix a bug once and it affects all subclasses
- If you understand the base class's interface, you can generally use any subclass easily
- Any function that can operate on a superclass can operate on a subclass w/o changes

Cons:

- Often results in poor encapsulation (derived class uses base class details)
- Changes to superclasses can break subclasses (Fragile Base Class problem)

Implementation Inheritance

- derived class inherits the method implementations of the base class, but NOT its public interface
- public interface of the base class is hidden from the outside world
  - often done with private or protected inheritance
- the derived class is NOT as subtype of the base class because they have different interfaces

```
// C++
class Collection {
public:
  void insert(int x) { ... }
  bool delete(int x) { ... }
  int count(int x) const { ... }
  ...
};

// C++
class Set: private Collection    // private inheritance
public:
  void add(int x)
    { if (howMany() == 0) insert(x); } // insert can be called within the class, but
not outisde
  bool erase(int x)
    { return delete(x); }
  bool contains(int x) const
    { return count(x) > 0; }
  ...
};
```

- Set *privately* inherits from Collection, it inherits all of its public/protected methods, but it does NOT expose the public interface (insert(), delete(), count(), ...) of the base Collection class publicly.

Implementation Inheritance Best Practices

- generally frowned upon
- if you want to have a class "inherit" all of the functionality of a base class, but hide the public interface, use composition + delegation
- why? using inheritance unnecessarily exposes the implementation details of base class to derived class -- dangerous!

Inheritance Mechanics

Construction

Every derived class constructor does two things:

- calls its superclass's constructor to initialize the immediate superclass part of the object
- initializes its own parts of the derived object
- In most languages, you must call the superclass constructor first from the derived class constructor, before performing any initialization of the derived object:

```java
// Construction with inheritance in Java
class Person {
 public Person(String name) {
   this.name = name;
 }
 private String name;
}

class Nerd extends Person {
  public Nerd(String name, int IQ) {
    super(name);  // calls superclass constructor first
    this.IQ = IQ; // then initializes members of the drived class
  }
  private int IQ;
}

class ComedianNerd extends Nerd {
  public ComedianNerd(String name, int IQ, String joke) {


  }
  private String joke;
}
```

- in other languages, like Swift, the derived constructor initializes the derived object's members first, then calls base class constructor.

- some languages will auto-call the base-class constructor if it's a default constructor (has no parameters), but other languages, like Python *always* require you to call the base constructor explicitly:

```
# Python inheritance+construction
class Nerd(Person):
  def __init__(self, IQ):
    super().__init__()  # if we leave this out, we simply won't initialize our Person base part!
    self.IQ = IQ
```

Destruction

- derived destructor runs its code first and then implicitly calls the destructor of its superclass
- repeats all the way to the base class

Finalization

- some languages require an explicit call from a derived class finalizer to the base class
- in other languages, the derived finalizer automatically/implicitly calls the base class finalizer

Overriding Methods

- in base class, you must designate which methods can safely be overridden and methods that shouldn't be overridden
- in some languages you declare explicitly which methods MAY be overridden
  - eg: `virtual` in C++

```
// C++ uses virtual to allow overriding
class Person {
public:
  string getName() const
   { ... }
  virtual void talk() const   // override
   { ... }
};

class Student: public Person {
public:
  virtual void talk() const override // virtual and override good style
    { /* OK to override! */ }
  ...
};
```

- in other languages you declare explicitly which methods MUST NOT be overridden.
  - eg: `final` in Java
  - `@final` decorator in Python indicates shouldn't override, but isn't enforced by interpreter
- it's good style to use *override indicators* to prevents bugs/make code readable!
  - it's easy to accidentally define an OVERLOADED version of a method rather than an OVERRIDING a method!

    - these also allow for compile time checks

```
class Person {
public:
  virtual void talk()
   { cout << "I'm a person!\n"; }
  ...
};

class Student: public Person {
public:
  virtual void talk(string to) override   // ERROR: doesn't override
  {
   cout << "Hi " << to <<
            " I'm a student!\n";
  }
};
```

- sometimes a derived version of a method needs to use a superclass implementation to get things done
    - C++: `SuperClass::method()`, can as many super classes up: `SuperSuperClass::method()`
    - Java: `super.method()`, can only call super of immediate super class
    - Python: you may use a class name OR a reference to super() to call a superclass method! `super().method()` or `SuperClass.method(self)`

Why restrict methods to only immediate super class?

- not doing this violates encapsulation, and allow a subclass to bypass its parent classes behavior
- derived class should only know about the most derived parent version's methods
- using methods from superclasses might cause behavioral issues in the immediate superclass, since it will be disintermediated from one or more calls.

Multiple Inheritance and Its Issues

- **Multiple inheritance** - a derived class inherits implementations from two or more superclasses
- considered an anti-pattern, and forbidden in most languages!
- In cases where you want to use multiple inheritance, have your class implement multiple interfaces instead.

Here's an example where there's a problem:

```
class CollegePerson {
public:
  CollegePerson(int income) { income_ = income; }
  int income() const { return income_; }
private:
    int income_;
};

class Student: public CollegePerson {
public:
  Student(int part_time_income) :
    CollegePerson(part_time_income)
  {/* Student-specific logic */ }
};

class Teacher: public CollegePerson {
public:
  Teacher(int teaching_income) :
    CollegePerson(teaching_income)
  { /* Teacher-specific logic */ }
};

class TA: public Student, public Teacher  {
public:
    TA(int part_time_income, int ta_income) :
      Student(part_time_income),
      Teacher(ta_income)
    { /* TA-specific logic */ }
};

int main() {
 TA amy(1000, 8000);
 cout << "Amy's pay is $"
      << amy.income();
}
```

- **diamond pattern** - when a derived class D inherits from a base class B twice by inheriting from two distinct classes X, Y that both on their own are derived from B
  - and thus has two copies of every member field, etc.
- both `Student` and `Teacher` inherit from the same base class `CollegePerson`
- so TA has TWO instances of the base class--TWO `income` fields and `income()` method--ambiguous which to use

Resolving Diamond pattern:

- Some languages, like C++ will generate an error for this: "error: non-static member income' found in multiple base-class subobjects of type 'CollegePerson'"
- Other languages have a mechanism to pick one of the two functions to always call, or perhaps let the programmer specify which base version to call (e.g., TA.income())

Other issues:

- What if two base classes define a method with the same prototype, and then a third class inherits from both base classes? When calling that method, which version should be used?

Abstract Methods and Classes

Abstract methods are methods that define an interface, but don't have an implementation. For instance, area() and perimeter() are abstract methods in the example below:

```
class Shape {
public:
  double area() const = 0;
  double perimeter() const = 0;
};
```

An abstract method defines "what" a method is supposed to do along with its inputs and outputs – but not "how." An abstract class is a class with at least one abstract method - it defines the public interface and partial implementation for subclasses.

When defining a base class, most languages allow the programmer to specify one or more "abstract methods", which are just function prototypes without implementations.

```
class Shape { // C++ class
public:
  // area and rotate are "abstract" methods
  virtual double area() const = 0;
  virtual void rotate(double angle) = 0;
  virtual string myName()
    { return "abstract shape"; }
};
```

```
abstract class Shape {  // Java class
 // area and rotate are "abstract" methods
 public abstract double area();
 public abstract void rotate(double angle);
 public String myName()
   { return "abstract shape"; }
};
```

Question: Can we instantiate an object of an A.B.C. like Shape?

```
int main() {
  Shape s;  // ?????
  cout << s.area(); // ????
}
```

Answer: You can't instantiate an object of an ABC because it has an incomplete implementation. What if the user tried to call s.area()? There's no method implementation to call!

So why define abstract methods at all? Why not just provide dummy implementations in the base class?

```
class Shape { // C++ class
public:
  // area is an "abstract" method
  virtual double area() const { return 0; }   // dummy version returns 0
  virtual void rotate(double angle) { }        // dummy version does nothing
  virtual string myName()
    { return "abstract shape"; }
};
```

Answer: An abstract method forces the programmer to redefine the method in a derived class, ensuring they don't accidentally use a dummy base implementation in a derived class!

OK what about abstract classes and types? Just as with concrete classes, when you define a new abstract class, it automatically creates a new type of the same name! More specifically, it defines a *reference type*, which can ONLY be used to define pointers, references, and object references... not concrete objects.

When should you use abstract methods/classes

Use abstract methods/classes under the following circumstances.

- Prefer an abstract method when there's no valid default implementation for the method in your base class.
- Prefer an abstract class over an interface when all subclasses share a common implementation (one or more concrete methods or fields), and you want the derived classes to inherit this implementation

In contrast, you would inherit/implement an interface when you have a "can-do" relationship, e.g., a Car can do Driving, so a car might implement a Drivable interface.

Inheritance and Typing

When we use inheritance to define classes, we automatically create new supertypes and subtypes! The base class (e.g., Person) defines a supertype. The derived class (e.g., Student) defines a subtype. And given the rules of typing, we can then use a subtype object anywhere a supertype is expected.

As we learned earlier, any time you define a class or an interface, it implicitly defines a new type.

The type associated with a concrete class is specifically called a value type. Value types can be used to define references, object references, pointers, and **instantiate objects**.

The type associated with an interface OR abstract class is called a reference type. Reference types can ONLY be used to define references, object references, and pointers.

Ok, so how does typing work with inheritance? Consider these two classes:

```
class Person {
public:
  virtual void talk()
    { cout << "I'm a person"; }
  virtual void listen()
    { cout << "What'd you say again?"; }
};

class Student: public Person {
public:
 virtual void talk()
   { cout << "I hate finals."; }
 virtual void study()
```

```
    { cout << "Learn, learn, learn"; }
};
```

Well... the Person base class definition implicitly defines a new type called Person. And the Student subclass definition implicitly defines a new type called Student. And since Student is a subclass of Person the Student type is a subtype of the Person type!

Ok, and how does typing work with interface inheritance? Consider this example:

```java
// Java interface inheritance
public interface Washable {
  public void wash();
  public void dry();
}

public class Car implements Washable {
  Car(String model) { ... }
  void accelerate(double acc) { ... }
  void brake(double decel) { ... }

  public void wash() {
   System.out.println("Soap & bucket");
  }
  public void dry() {
    System.out.println("Dry with rags");
  }
}
```

Well... the Washable interface definition implicitly defines a new type called Washable. And the Car class definition implicitly defines a new type called Car. And since Car is an implementer of Washable the Car type is a subtype of the Washable type!

Inheritance and Typing Challenges

Question: When would a derived class not be a subtype of its base class? Answer: When we privately inherit from a base class, the derived class is NOT a subtype of the superclass's type, nor does it share the interface of the base class!

Question: When could a class have a supertype, but not a superclass? Answer: A class could have a supertype of an interface type, without having a superclass, since an interface is not a class (at least in some languages).

Subtype Polymorphism

- **Subtype Polymorphism** - the ability to substitute an object of a subtype anywhere a supertype is expected
  - only in statically typed languages
  - don't have types for variables in dynamically typed languages
- More technically, S.P. is where code designed to operate on an object of type T operates on an object that is a subtype of T.
- as learned in datapalooza, typing rules state that if a function can operate on a supertype, it can also operate on a subtype!

- same holds true for supertypes and subtypes associated with inherited classes! And for classes that implement interfaces!

Also works because the class associated with the subtype supports the same public interface as the class associated with the supertype.

```
class Washable { // Interface
  virtual void wash() = 0;
  virtual void dry() = 0;
};
class Car: public Washable { ... }

void washAThing(Washable& w) {  // this function  accepts a supertype
  w.wash();
}

void cleaningCrew() {
  Car Subaru;
  washAThing(subaru);  // we're passing a subtype object
}
```

We also expect the methods of the subclass to have the same *semantics* as those in superclass.

```
class Shape {
public:
  virtual double getRotationAngle() const {
      // returns angle in radians
  }
};

class Square: public Shape {
public:
  virtual double getRotationAngle() const {
    // returns angle in degrees
  }
};
```

- **Liskov Substitution Principle** - subclass adheres to the interface and semantics of the base class
- so we can truly substitute any subtype object for a supertype

Subtype Polymorphism in Dynamically-typed Languages?

- Subtype polymorphism occurs when we process a subtype object as if it were an object of the supertype, but in dynamically-typed languages, variables have NO type at all! so no subtype polymorphism
- this code uses duck-typing, not subtype polymorphism

```
class Car:
  def speedUp(self, accel):
    ...

class HybridCar(Car):
  def speedUp(self, accel):
    ...

def drive_car(c):
  c.speedUp(5)

def main():
  prius = HybridCar()
  drive_car(prius)  # duck-typing
```

Dynamic Dispatch

```
void foo(Shape& var) { // c++
  cout << var.area();
}
```

- **dynamic dispatch** - the process of figuring out which method to call when a call is made to a virtual method at runtime
- when you call a method on var, the actual method that gets called is determined at run-time based on the target object that var refers to. The determination is made in one of two ways:
  - Based on the target object's class, or
  - By seeing if the target object has a matching method (regardless of the object's type or class), and calling it if one was found - in languages without classes and just objects

Dynamic Dispatch in Statically-typed Languages

- language examines the class of an object at the time of a method call and uses this to "dynamically dispatch" to the class's proper method
- Typically implemented by adding a hidden pointer into each object which points at a **vtable** for the class
- **vtable** - an array of function pointers which points to the proper method implementation to use for the current class
- every class has its own vtable and only virtual methods are included
- Dynamic dispatch is NOT used to call regular instance or class methods
- a standard function call is used in these cases--called **static dispatch** since the proper function is determined at compile time (statically)

Performance: Static vs Dynamic Dispatch

- dynamic dispatch is slower than static dispatch, since we need to look up function pointers in the vtable at runtime before performing the call
- Static dispatch calls are hard-wired at compile time into the generated machine code, and thus much faster (a direct machine language call).

Dynamic Dispatch in Dynamically-typed Languages

- a class may define a default set of methods for its objects, but the programmer can add/remove methods at runtime to classes or sometimes even individual objects!
- So when a method is called, the language can't necessarily rely upon an object's class or type to figure out what method to use!
- to do this, the language stores a unique vtable in every object!

```javascript
// Javascript dynamic dispatch example
function makeThemTalk(p) {
  console.log(p.name + " says: " + p.talk());
}

var person = {
  talk: function () {
    return "Java is meh";
  },
  change: function () {
    this["talk"] = () => {
      // change talk function!
      return "I <3 Smalltalk";
    };
  },
};

makeThemTalk(person); // Java is meh
person.change();
makeThemTalk(person); // I <3 Smalltalk
```

Subtype Polymorphism vs Dynamic Dispatch

- Subtype polymorphism is a statically-typed language concept that allows you to substitute/use a subtype object anywhere code expects a supertype object.
- Since the subtype class shares the same public interfaces as the supertype, the compiler knows that the types are compatible and substitution is allowed.
- dynamic dispatch is all about determining where to find the right method to call at run-time for virtual/overridable methods
- Dynamic dispatch occurs whether or not we have subtype polymorphism, as we've seen in dynamically-typed languages like Python.